AD-A093 264 VIRGINIA POLYTECHNIC INST AND STATE UNIV WASHINGTON --ETC F/G 9/2
SIMULA (TRADEMARK) FILE CLASSES.(U)
OCT 80 R J ORGASS AFOSR-79-0021
UNCLASSIFIED VPI/SU-TM-80-6 AFOSR-TR-80-1293 NL

1 OF 1
AD A
093264

END
DATE
FILMED
1-81
DTIC

EXTENSION DIVISION

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE
GRADUATE PROGRAM IN NORTHERN VIRGINIA

P. O. Box 17186
Washington, D. C. 20041
(703) 471-4600

SIMULA* File Classes†§

Richard J. Orgass

Technical Memorandum No. 80-6

October 8, 1980

[redacted]

Abstract

Two SIMULA class. *in_file* and *out_file* which are designed as replacements for the system defined classes ***Infile*** and ***Outfile*** in IBM SIMULA are described. These replacements provide a number of campabilities that make it easier to work with a terminal in interactive programs and provide for considerably improved error recovery.

* SIMULA is a registered trademark of the Norwegian Computing Center.

† Research Sponsored by the Air Force Office of Scientific Research, Air Force Systems Command, under Grant No. AFOSR-79-0021 The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

§ The information in this document is subject to change without notice The author, Virginia Polytechnic Institute and State University, the Commonwealth of Virginia and the United States Government assume no responsibility for errors that may be present in this document.

SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

(19) REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER (18) AFOSR-TR- 80 - 1293	2. GOVT ACCESSION NO. AD-A093264	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) SIMULA FILE CLASSES. (9) Technical (14) VPI/SU-TM-80-[illegible]		5. TYPE OF REPORT & PERIOD COVERED INTERIM
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Richard J. Orgass (12) 49		8. CONTRACT OR GRANT NUMBER(s) (15) AFOSR-79-0021
9. PERFORMING ORGANIZATION NAME AND ADDRESS Dept. of Computer Science Virginia Polytechnic Institute and State University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61102F (16) 2304/A5 (17) A5
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, D. C. 20332		12. REPORT DATE (11) 8 October 1980
		13. NUMBER OF PAGES 49
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) [illegible]		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Accession For	
NTIS GRA&I	✓
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Two SIMULA Class in file and out file which are designed as replacements for the system defined classes infile and Outfile in IBM SIMULA are described. These replacements provide a number of campabilities that make it easier to work with a terminal in interactive programs and provide for considerably improved error recovery.

411 731

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

SIMULA File Classes

The system defined file classes provided with IBM SIMULA are designed for a batch environment and there are many obstacles to using these file classes to create interactive programs in the VM/CMS environment. For example, it is necessary to use DD names when creating an instance of a file class and this requires issuing filedef commands before program execution begins. When a terminal user responds to a prompt by simply entering <CR>, this response is interpreted as an end-of-file and any further attempt to read from the terminal causes an error termination. The error recovery provided by the system defined file classes is quite simple and direct: execution is terminated. In an interactive environment one expects to provide reasonable opportunities for a user to recover from minor errors.

In an interactive environment it is often very useful to be able to create log files which contain all of the user's input and to be able to read such files in place of terminal input so as to recover from either human errors or other misadventures. When such log files are being read to recreate the state of a program, it is desirable to recreate the terminal transcript for the user so that the user is aware of the state of the program when he is finally in a position to continue work from the point where he was interrupted.

This report describes two SIMULA classes. *in_file* and *out_file* which provide for dynamic file naming, error recovery, log file creation and application as well as character translation for use with APL terminals.

The file classes described here have all of the attributes of the system defined file classes as well as additional attributes that are described here. It is assumed that the reader is familiar with the system defined file classes and no further documentation of the attributes of these classes is provided.

These file classes provide error recovery for both user errors and for program errors. There are a number of coding errors that may be corrected during program execution without changing the source text and this kind of error recovery requires some justification. The CMS SIMULA environment is basically the result of grafting a batch processing run time environment into CMS. As a result. the only error action taken by the system is to terminate execution and (optionally) print either a symbolic or hex dump. As a result, a trivial coding error can require another compilation of the source program only to detect another error. The coding error recovery was provided to improve the efficiency of program debugging. If a symbolic debugger were available. many of these corrections would be inappropriate because they could be dealt with by the debugger. Thus. this error correction should be viewed as a (hopefully) temporary program development tool for use until a symbolic debugger is available.

Section 1 describes the use of class *in_file* and Section 2 describes the use of class *out_file* by means of examples. Section 3 provides instructions for using these file classes on the Virginia Tech VM/CMS system. A listing of the source text of these the code that is added to a user program appears as an appendix.

The following conventions are used in this report. When a sketch of program text appears. this text is printed in Algol publication format. i.e., **boldface** keywords, *italic* identifiers and *Italic Edit Case* for system defined identifiers. The reader should assume that this is an example that was constructed to explain a particular concept rather than an excerpt from a specific program. When an extract of an actual program occurs in the document, it is printed in a `fixed pitch` font with upper case keywords, Edit Case system defined identifiers and lower case user defined identifiers. System commands and input text to be typed by a user also appear in a `fixed pitch` font.

1. Input Files.

Just as for the system defined class *Infile*, the first step in reading a file is to create an instance of a file object. This might be done as follows:

```
ref(in_file) input;
...
input :- new in_file(<file spec>);
```

In this code fragment, <file spec> is a CMS file id or the string `tty:` in any mixture of upper and lower case letters. If a portion of the file id is missing, it is provided in accord with the defaults described below. When this statement is executed, the value of *input* is a closed file object whose input will be received from the device specified by <file spec>.

If the actual parameter of *in_file* is the text object `TTY:`, then input is read from the terminal. Note that the text objects `tty:`, `tTy:` and so forth are equivalent in this context. If the actual parameter is a text object other than one of these strings, then the actual parameter is interpreted as a (partially specified) CMS file id.

Recall that a file id is a string of the form <fn>[<ft>[<fm>]]. If the <ft> is omitted, the <ft> DATA will be used. If <fm> is omitted, all accessable disks will be searched in accord with the search list specified by the profile. The first file in this search that matches <fn> <ft> will be selected. If there is no such file, a message will be printed on the terminal and the user will be asked to specify another file id. By replying to this request for a new file id with the string `CMS:`, the user may enter CMS subset to examine his directory to select an input file. To return to the running program, the user enters the string `RETURN` (in any mixture of upper and lower case letters).

A program segment to read a file id from the terminal and then create an *in_file* might be the following:

```
REF(in_file) input;
TEXT file_id;
   ...
Outtext("Fileid: ");
Breakoutimage;
Inimage;
file_id :- Copy(Image.Strip);
input :- NEW in_file(file_id);
```

and the terminal transcript would look like this (assuming that the file MUMBLE FOO does not exist):

```
Fileid: mumble foo
? File MUMBLE FOO does not exist.
Enter file id: /CMS:/
```

At this point, the user could enter CMS subset and look around to select the appropriate file. When this is completed, the user enters `RETURN` and the dialog continues as follows (assuming that the actual input file is ALPHA SIMULA):

```
# return
[Returning to SIMULA program execution.]
Enter file id: /CMS:/: alpha simula
```

After this, the value of *input* is a closed input file object connected to file ALPHA SIMULA. Observe that the prompt character is # when CMS subset is being used.

Instances of class *in_file* are opened in the same way that instances of the system defined class *Infile* are opened, i.e., to open the *in_file* object *input*, the statement

```
input.Open(Blanks(80))
```

is executed. There is, however much more run time error recovery when the *Open* attrtibute of *in_file* is used. The following error recovery is provided:

(1) If the file object associated with *input* is already open, then an error message is printed on the terminal and the user is given the option of continuing execution or terminating execution. The text of the message is:

```
? in_file.Open: File ALPHA SIMULA is already open.
Do you want to continue reading the file? /y/:
```

If the user responds with a carriage return, execution continues as if the file were closed when open was called. However, if records have already been read from the file, reading continues from the next record; the file is not rewound.

(2) If the record length of the file is greater than the length of the actual parameter to the file object, the user is given the option of extending the *Image* attribute of the file object or terminating execution. For example, if the above call on *Open* were executed and if the record length of file ALPHA SIMULA is 132, the following terminal transcript would occur:

```
? in_file.Open: File ALPHA SIMULA has a record length greater
than the length of the provided buffer.
Do you want to extend the buffer? /y/:
```

If the user answers yes by entering a return, then *input.Image* is extended to *Blanks(132)* and if the user answers no, then execution is terminated.

Class *in_file* has the usual attributes *Setpos*, *Pos*, *More*, *Length*, *Close*, *Inimage*, *Lastitem*, *Intext*, *Inint*, *Infrac*, *Inreal* and *Inchar* of the system defined class *Infile* but these attributes of the class provide for error recovery. In the system defined class, it is a run time error to call one of these procedures when the file associated with the object is closed. When class *in_file* is used, an attempt to read from a closed file results in an error message and the user may decide to open the file or terminate execution. For example, if *input.Inint* were called while file ALPHA SIMULA is closed, the following dialog would occur:

```
% in_file.Inint: File ALPHA SIMULA is closed.
Do you want to open this file? /y/:
```

If the user answers with a return or yes, then the file is opened and the first record is read to satisfy the call to *Inint* and if the user answers no then execution is terminated after an optional symbolic dump.

The attribute *Close* of *in_file* behaves in the same way as the system defined attribute of *Infile* except that a call to *Close* will not cause a termination of execution if the file associated with the object is already closed. Instead, the following advisory messages are printed on the terminal:

```
[% in_file.Close: File ALPHA SIMULA is already closed.]
[Execution continues.]
```

The attribute *Endfile* of class *in_file* behave in exactly the same way as the corresponding attribute of the system defined class *Infile*.

Instances of class *in_file* have additional attributes that are not possesed by instances of the system defined class *Infile*. Each of these attributes is described below. The attributes are described by exhibiting the declaration of the attribute followed by a description of the attribute.

boolean procedure *Opened*

This procedure returns the value **true** when the file associated with the instance of class *in_file* is open and **false** otherwise. This attribute duplicates an attribute of the file classes in DEC-10 SIMULA. It is sometimes convenient to use this attribute when one wishes to take a different course of action depending on the state of the file.

text procedure *file_spec*

The return value of this procedure is a text object whose value is the CMS file id of the file associated with the instance of class *in_file*. The text object is in the same format as the output of the CMS command LIST. That is, the file name and file type each occupy eight spaces (possibly padded with blanks on the right) and the file mode occupies two spaces. There is a single space between the file name (padded to eight spaces) and the file type and another space between the file type (padded to eight spaces) and the file mode.

boolean *echo*

This attribute of an *in_file* is meaningful only if the file associated with the instance of the class is a disk file. When the file is a disk file, the value of *echo* controls the behavior of the file object as follows: (1) If *echo* has the value **true**, then characters read from the file are copied onto the terminal as they are being read from the disk. (2) If *echo* has the value **false** then this echoing is not performed.

This attribute of class *in_file* was included to make it possible to read files which contain user input that was typed during a previous execution of the program and then reproduce the terminal transcript in the continuation session. The *echo* attribute combined with the *log* attribute described below makes it very easy to write code to function in this way. See the example below.

ref(*out_file*) *log*

If the value of this attribut of an *in_file* is **set to none**, then the class instance behaves just as one would expect. However, the following occurs if *log* is different from **none**:

Each line of input from the input file (or terminal) is written to the file associated with the value of *log*. Note that **only** the input received from the device is written to this file; program output is **not** recorded.

boolean *divert*

The value of this attribute controls the action of class *in_file* when an end-of-file is encountered in a disk file. If the value of *divert* is **false**, then the instance of class *in_file* behaves as usual upon reaching the end of file. On the other hand, if *divert* is **true**, then and end-of-file from the disk is used to transfer the source of input from the disk file to the terminal. For example, suppose that a program is reading file ALPHA SIMULA and that the end-of-file is encountered while *divert* is **true**. The terminal transcript would look like this:

```
[% in_file.Inimage: Input from file ALPHA SIMULA diverted to
TTY:.]
```

After this, additional reads from the file (including the read which initiated the end-of-file) are satisfied by reads from the terminal. This attribute is particularly useful when reading log files generated with the help of the *log* attribute described above.

Example

The following code fragment illustrates the use of class *in_file* to create an input file which uses the echoing and logging facilities described above.

```
ref(in_file) input;
ref(out_file) log_file;

  . . .
Outtext("Input file: ");
Breakoutimage;
input :- new in_file(Image.strip);
Outtext("Log file: ");
Breakoutimage;
Inimage;
log_file :- new out_file(Image.Strip);
  . . .
input.echo := true;
input.divert := true;
input.log :- log_file;
```

In the first sequence of executable statements, an *in_file* and an *out_file* are created using files specified by the user from the terminal. In the second group of executable statements, the *echo*, *divert* and *log* attributes of *input* are initialized. The files must, of course, be opened befor reading and writing begins.

When this program is executed, the user is first asked to specify the input file from which data is to be read. If there is no existing log file, then the appropriate response would be the string `tty:` to cause all input to be read directly from the terminal. The file id that is the response to the prompt `Log file:` becomes the file id of a file that contains all input that was entered from the terminal in response to calls to *input.Inimage*. At some point in program execution, it might be desirable to suspend execution of the program. This can be accomplished by entering control-C (↑C) in response to a request for input. Suppose that the log file that was created during this execution was file ALPHA1 LOG.

The next time the program is executed, the user could answer ALPHA1 LOG to the prompt `Input file:`. The response to the prompt for a log file might be ALPHA2 LOG. During this execution of the program, all of the responses from the first execution will be read from file ALPHA1 LOG and both these responses and program output would be printed on the terminal. When the end of file ALPHA1 LOG is encountered, a message will be written on the terminal and additional inputs will be read from the terminal. At the end of this program execution, file ALPHA2 LOG will contain all of the user responses from the first and second execution of the program. This file could then be used to bring the program back to its state at t..e end of the second execution. After this, execution could continue as though the entire computation had been done in one session.

Using log files provides a certain amount of security agains accidential loss of important results and provides a rather limited replacement for a checkpoint facility (which is not available in CMS).

Useful Details

The usual CMS convention is that an empty line from the terminal is an end-of-file for the terminal. This convention proves to be quite inconveneint in an interactive environmnet. When using class *in_file*. the following conventions apply:

(1) An empty line, i.e., responding to a request for input by simply hitting the return key, is read as an empty line consisting of *Image.Length* blanks.

(2) A line consisting of control-Z (↑Z) followed by return is read as an end-of-file from the terminal.

(3) A line consisting of control-C (↑C) followed by return is read as a request to terminate execution. Execution is terminated and all open files are closed.

Note that in (2) and (3) the input line must consist of the control character immediately followed by return. A control character followed by a blank and then a return will NOT be interpreted in this way -- it will be transmitted to the program.

This set of typing conventions was implemented to provide a more reasonable terminal interface for CMS SIMULA. It is still not a satisfactory solution to the problem of providing a comfortable interactive environment but it is substantially more convenient than the version provided by the system.

2. Output Files.

Just as for the system defined class *Outfile*. the first step in writing a file is to create an instance of a file object. This might be done as follows:

```
ref(out_file) output;
. . .
output :- new out_file(<file spec>);
```

Just as for an input file object. in this code fragment <file spec> is a CMS file id or the string `tty:` in any mixture of upper and lower case letters. If a portion of the file id is missing, it is provided in accord with the defaults described below. When this statement is executed. the value of *output* is a closed file object whose output will be directed to the device specified by <file spec>. [An obscure detail possibly of interest to hackers: A filedef for the output file is not executed during the creation of an *out_file* because the record length of the file is unknown until the call to *Open* which must occur before data transmission.]

If the actual parameter of *out_file* is the text object `TTY:`, then output is written to the terminal. Note that the text objects `tty:`, `tTy:` and so forth are equivalent in this context. If the actual parameter is a text object other than one of theses strings, then the actual parameter is interpreted as a (partially specified) CMS file id.

Recall that a file id is a string of the form <fn>[<ft>[<fm>]]. If the <ft> is omitted, the <ft> LOG will be used. If <fm> is omitted, then the mode will be determined by examining all accessable disks in the search list specified by the profile. The file mode will be the mode of the first disk in this sequence with the property that the user has write access to the file. In most applications, this will be the A(191) disk. If the user does not have write access to any disk, a non-recoverable error will occur at the first attempt to transfer data to the file.

A program segment to read a filed id from the terminal and then create an *out_file* might be the following:

```
REF(out_file) output;
TEXT file_id;
  . . .
Outtext("Output file: ");
Breakoutimage;
file_id :- Copy(Sysin.Image.Strip);
output :- NEW out_file(file_id);
```

and the terminal transcript would look like this:

```
Output file: fumble foo
```

After this, the value of *output* is a closed file object that will write output to disk file FUMBLE FOO. The next step is to open the file with the procedure call:

```
output.Open(Blanks(132))
```

Executing this call will open the file named in the creation of *output* (FUMBLE FOO in this case) to be written as a fixed length record file with a record length of 132 characters. As in the system defined class *Outfile*, the length of the parameter of *Open* determines the record length of the file to be written.

The system defined class *Outfile* will terminate execution if the file is open when a call to *Open* is executed. The *Open* attribute of an *out_file* will give the user the option of continuing execution in this case. The terminal transcript lookis like this:

```
? out_file.Open: File FUMBLE FOO is already open.
Do you want to continue writing the file? /y/:
```

If the user selects the answer "yes" by hitting the return key, then data transmission to the file continues. On the other hand, if the user responds with the character n (in upper or lower case), execution will be terminated. If the user elects to continue program execution, additional outputs written to the file will be appended to the data already written to the file.

Class *out_file* has the usual attributes *Setpos*, *Pos*, *More*, *Length*, *Outchar*, *Outint*, *Outrac*, *Outreal*, *Outfix* and *Outimage* of class *Outfile*. However, there is provision for error recovery if there is an attempt to execute one of these procedures when the file is closed. Here is an example of the terminal dialog:

```
% out_file.Setpos: File FUMBLE FOO is closed.
Do you want to open this file? /y/:
```

If th. user answers this query affirmatively, then the program will request the record length of the output file as follows:

```
Enter length of output lines: /80/:
```

The response to this prompt may be any positive integer *less than* 256 (a VM imposed limitation). The file associated with the file object will be opened to write with the specified record length. Note that this will cause the destruction of data that is already in the file!

The attribute *Outtext* of class *out_file* performs in the same way as the system defined version but it has two error correcting capabilities. As for all of the other attributres of the class, it is possible to recover from an attempt to write on a closed file. The second extension provides for recovery from a fairly common error. In the system defined version, if *Outtext* is called to write a text object consisting of, say, 25 characters and there is only space for 24 characters in *Image*, then an error termination will occur. In the version described here, the first 24 characters of the argument to *Outtext* will be added to *Image* and then a call to *Outimage* is executed. After this call, the remaining character in the parameter of *Outtext* is placed into *Image*. More generally, a text object of any length can be transmitted to an output file with a single call to *Outtext*. The text object will be written to the file using as many lines as are needed to contain the text value. If less than a full line of characters remain after lines are written, these characters remain in *Image* and *Image.Pos* has the appropriate value.

The attribute *Close* of class *out_file* provides for error recovery if the procedure is called when the file is already closed. Here is an example of the message that is printed on the terminal when this occurs:

```
[% out_file.Close: File FUMBLE FOO is already closed.]
[Execution continues.]
```

Class *out_file* has a number of additional attributes that provide for printing prompts and reading a response on the same line, for echoing output to a disk file onto the terminal and for performing translation so that output can be written to APL terminals while preserving the characters in the text. These attributes are described by exhibiting the declaration of the attribute followed by a description of the attribute.

procedure *Breakoutimage*

When the procedure *Outimage* is called to write a record to a disk file, the value of the *Image* attribute (including trailing blanks) is written to the file. In contrast, when *Outimage* is called to write a record to the terminal, the value of *Image.Strip* followed by <CR><LF> is written to the terminal. This means that the next piece of input or output will appear on the next line. In many interactive applications, it is convenient to be able to transmit characters to the terminal without a trailing <CR><LF>. In DEC-10 SIMULA, the attribute *Breakoutimage* of class *Outfile* provides this capability. The corresponding attribute of class *out_file* has the same property for IBM SIMULA.

The *Breakoutimage* attribute of an *out_file* may be described as **follows**: If the instance of *out_file* is associated with the terminal, then *Image.Sub(1,Image.Pos)* is written to the terminal without a trailing <CR><LF>. If the instance of *out_file* is associated with a disk file, then *Breakoutimage* is equivalent to *Outimage*.

Some examples might illustrate an application of *Breakoutimage*. Suppose the program fragment:

```
        Outtext("Enter your name: ");
        Outimage;
        Inimage
```

is executed. The terminal transcript might look like this:

```
        Enter your name:
        Bill Jones
```

Notice that the prompt and response are on different lines. On the other hand, if the program fragment:

```
        Outtext("Enter your name: ");
        Breakoutimage;
        Inimage
```

were executed, the terminal transcript would look like this:

```
        Enter your name: Bill Jones
```

boolean procedure *Opened*

This procedure returns the value **true** if the file associated with the file object is open and **false** otherwise. This attribute of class *out_file* corresponds to the attribute of class *Outfile* in DEC-10 SIMULA with the same name. This attribute makes it easier to write programs that take different courses of action depending on the state of a particular file

text procedure *file_spec*

The return value of this procedure is a text object whose value is the CMS file id of the file associated with the instance of class *out_file*. The text object is in the same format as the output of the CMS command list. That is, the file name and file type each occupy eight spaces (possibly padded with blanks on the right) and the file mode occupies two spaces. There is a single space between the file name (padded to eight spaces) and the file type and another space between the file type (padded to eight spaces) and the file mode

boolean *echo*

The attribute *echo* is only meaningful if the instance of class *out_file* is associated with a disk file. In this case, if *echo* has the value **true**, then all lines written to the file are also written to the terminal. If *echo* is **false**, then this echoing to the terminal is not performed. If the instance of *out_file* is associated with the terminal, the value of *echo* is ignored.

APL Character Set Support

ASCII/APL terminals have the property that they can display characters in either the ASCII or APL character sets. Most modern ASCII/APL terminals also have the property that the character set can be changed under the control of the host computer to which the terminal is conneced. This second character set on a terminal provides a number of exciting possibilities for the use of terminals but it also imposes some additional work on the programmer.

For example, if a program writes an ASCII string to the terminal when the terminal is in the APL character set, it is quite likely that the text printed on the terminal will be unintelligible to the user. A similar situation arises if a string of characters in the APL character set is sent to the terminal when the terminal is in the ASCII character set. Thus, it is the responsibility of the programmer to keep track of the character set state of the terminal and to take appropriate action.

The use of ASCII/APL terminals is further complicated by the fact that there are two generally accepted mappings of APL characters onto ASCII characters. This unfortunate state of affairs is a result of posponing a decision concerning a standard for this mapping until terminal manufactures were committed to both character mappings. The two mappings of APL characters onto ASCII characters are called *key-paired* and *bit-paired*.

Class *out_file* has six procedure attributes that make it easier to work with ASCII/APL terminals. These include procedures to change between the three character sets (ASCII, key-paired APL and bit-paired APL) and to determine the state of the terminal character set. In addition, there are procedures to write output while bypassing the character set control and translation mechanisms.

By definition, character set 0 is the ASCII character set, character set 1 is key-paired APL and character set 2 is bit-paired APL

integer procedure *term_type*

The return value of this procedure is the character set number of the current terminal character set. When the return value of *term_type* is 0, text written to the terminal or a disk file is transmitted without translation.

When the return value of *term_type* is 1 then all output transmitted to the terminal or disk file by way of calls to *Outimage* or *Breakoutimage* is translated to preserve the appearance of the text object when it is printed on a key-paired ASCII/APL terminal. This translation converts lower case letters into the (upper case) APL letter and converts upper case letters into underscored APL letters. The graphics are translated subject to preserving the appearance or function of the character.

When the return value of *term_type* is 2 then all output transmitted to the terminal or disk file by way of calls to *Outimage* or *Breakoutimage* is translated to preserve the appearance of the text object when it is printed on a bit-paired ASCII/APL terminal. The translation is as described above.

When an instance of *out_file* is created, the defalut terminal character set is 0 and this will be the return value of *term_type*.

procedure *set_ascii*

When this procedure is called, the character set associated with the file is changed to ASCII. If the character set was different from ASCII and if there were characters in *Image*, then these characters are emitted in the previous character set. Characters to change the character set of the terminal are sent to the terminal or file.

procedure *set_key_paired*

When this procedure is called, the character set associated with the file is changed to key-paired APL. The same character set change and buffer emptying described above is performed.

procedure *set_bit_paired*

When this procedure is called, the character set associated with the file is changed to bit-paired APL. The character set change characters and buffer emptying procedure is similar to the procedure described for *set_ascii*.

procedure *apl_outimage*

This procedure is the same as *Outimage* except that character translation as described above is not performed independent of the state of the terminal, disk file and character set.

procedure *apl_breakoutimage*

This procedure is the same as *Breakoutimage* except that character translation as described above is not performed independent of the state of the terminal, disk file and character set.

3. Directions for Using these File Classes

The file classes described here are implemented as declared classes which also make calls on procedures that are contained in LIBSIM. This section provides directions for incorporating these file classes into programs and a sketch of their implementation.

The files that support the file classes described here are stored on the 191 disk of CMS userid CSDULLES. If you are going to use these file classes, it's a good idea to add the following three lines to your PROFILE EXEC:

```
cp link csdulles 191 330 rr all
access 330 b/a
global txtlib libsim simlib
```

In any case, these three commands must be executed before you attempt to compile or execute a program that uses these file classes.

A program that uses these file classes must be compiled using the SIM exec that is provided on CSDULLES. Suppose that a program that uses these file classes is in file ALPHA SIMULA. This program would be compiled, loaded and executed by entering the CMS command:

```
sim alpha
```

When a program is compiled in this way, the usual default value of the compiler parameters are used. The output from the compiler (a text file) is loaded and then a module file is written to the disk. After the module is written, execution of the program begins. [It takes *the loader a terribly long time to load SIMULA programs and it is possible to save a great deal* of time by creating module files.] When this exec is invoked to compile a program, it will delete an existing module file with the same name.

It is necessary to make a few minor changes in a source program that uses these file classes. Suppose that the text of an ordinary SIMULA program consists of **begin** ... **end**. This program would be modified as follows:

```
%COPY PREFIX
BEGIN
  .
  .
  .
END;
%COPY POSTFIX
```

The effect of this is to incorporate the declarations of the classes *in_file* and *out_file* into the program. In addition. *Sysin* is declared and opened as an *in_file* and *Sysout* is declared and opened as an *out_file*. Observe that the **end** that terminates the program must be follwed by a sem-colon.

If you are interested in exploring the details of the code introduced into your program by this change. you might want to consult the listings of files PREFIX SIMULA and POSTFIX SIMULA which appear in the appendix. These files are added to your program by the two %COPY commands to the compiler.

Appendix

Listing of Source Code

The source code contained in files PREFIX SIMULA and POSTFIX SIMULA is shown on the following pages. This is a reproduction of the actual files as printed on a Diablo terminal using the utility program TTYSPL.

```
PREFIX   SIMULA    B1

%NOSOURCE
BEGIN
    EXTERNAL PROCEDURE  initial,
                        Abort;

    EXTERNAL ASSEMBLY PROCEDURE vpisave,
                                vpirest;
```

```
non-terminal devices.  If divert is TRUE, then when an
end-of-file is recognized the input stream is diverted
to the terminal and a message is printed on the
terminal.  This replaces the usual end-of-file recovery.
;
```

```
COMMENT Class in_file has all of the attributes of the system
        defined class Infile as well as additional attributes.
        The system defined attributes as implemented here include
        a substantial amount of interactive error recovery that
        is not in the system version.  The additional attributes
        are procedures or variables as follows.

        TEXT PROCEDURE file_spec

        The return value is the file specification associated
        with the instance of in_file.  See TM 79-8 for a
        description of file specifications.

        BOOLEAN PROCEDURE Opened

        The return value is TRUE if the file associated with
        the instance of in_file is open and FALSE otherwise.

        INTEGER PROCEDURE term_type

        The return value of this procedure is an integer
        that corresponds to the character set of the input
        device as follows:

                0       EBCDIC file or ASCII terminal
                1       Key-paired APL file or terminal
                2       Bit-paired APL file or terminal
                3       APL print train file

        ECHO

        This boolean variable is used together with input
        from non-terminal devices.  If echo is TRUE, then
        each input line is transmitted to the terminal as
        it is read from the device.  If echo is FALSE (the
        default value), this copying is not performed.  The
        ability to echo input to the terminal is useful when
        part of a session uses a previously created input
        file.

        DIVERT

        This boolean variable is used together with input from
```

```
BEGIN

    REF(Infile) f;

    REF(out_file) log;

    TEXT Image,
          ddname,
          message;

    BOOLEAN f_opened,
             f_endfile,
             divert,
             echo;

    INTEGER t_type,
             lrecl;

    EXTERNAL ASSEMBLY PROCEDURE  vpiboi,
                                  vpimoi,
                                  symbdump,
                                  vpiabort,
                                  vpiini;


    EXTERNAL BOOLEAN PROCEDURE   booleanrequest;


    EXTERNAL TEXT PROCEDURE     frontstrip,
                                upcase,
                                conc2,
                                getddinput;
```

```
PROCEDURE check_open(caller); VALUE caller; TEXT caller;
    IF NOT f_opened
         THEN BEGIN
                  message :- conc2("% in_file.",caller);
                  message :- conc2(message,": File ´");
                  message :- conc2(message,f_spec);
                  message :- conc2(message,"´ is closed.");
                  vpimoi(message);
                  IF booleanrequest("Do you want to open this file?",
                                    TRUE, TRUE)
                      THEN Open(Blanks(lrecl))
                      ELSE Abort("Attempt to access closed file.");
              END of check_open;


PROCEDURE eof_abort(source,file);
        VALUE source, file; TEXT source, file;
BEGIN
    EXTERNAL ASSEMBLY PROCEDURE vpiabort,
                                symbdump;

    message :- conc2("? in_file.",source);
    message :- conc2(message,": EOF while reading file ´");
    message :- conc2(message,f_spec);
    message :- conc2(message,"´.");
    vpimoi(message);
    IF booleanrequest ("Do you want to read the file again?",
                                FALSE, TRUE)
         THEN BEGIN
                   Close;
                   Open (Blanks(Image.Length))
              END
        ELSE Abort("Reading past end of file.");
END of eof_abort;
```

```
PREFIX   SIMULA    B1

BOOLEAN PROCEDURE opened;
    opened := f_opened;


TEXT PROCEDURE file_spec;
BEGIN
    check_open("file_spec");
    file_spec :- Copy(f_spec);
END of file_spec;


PROCEDURE Setpos(i); INTEGER i;
BEGIN
    check_open("Setpos");
    Image.Setpos(i);
END of Setpos;


INTEGER PROCEDURE Pos;
BEGIN
    check_open("Pos");
    Pos := Image.Pos
END of Pos;


BOOLEAN PROCEDURE More;
BEGIN
    check_open("More");
    More := Image.More
END of More;


INTEGER PROCEDURE Length;
BEGIN
    check_open("Length");
    Length := Image.Length
END of Length;
```

```
PROCEDURE Open(buf); TEXT buf;
BEGIN
    IF f_opened
        THEN BEGIN
                message :- conc2("? in_file.Open: File ´",f_spec);
                message :- conc2(message,"´ is already open.");
                vpimoi(message);
                IF NOT booleanrequest(
                        "Do you want to continue reading the file?",
                         TRUE, TRUE)
                        THEN Abort("Opening an open in_file.")
                        ELSE GO TO exit;
             END;
    IF (buf.Length < lrecl) AND (f_spec NE "TTY:")
        THEN BEGIN
                message :- conc2("? in_file.Open: File ´",f_spec);
                message :- conc2(message,"´ has a record length greater ");
                message :- conc2(message,"than the length of the provided ");
                message :- conc2(message,"bufrer."); vpimoi(message);
                IF booleanrequest(
                        "Do you want to extend the buffer?",
                        TRUE, TRUE)
                    THEN buf :- Blanks(lrecl)
                    ELSE Abort("LRECL of file greater than Image.Length.");
              END;

    IF (buf.Length < lrecl) AND (f_spec = "TTY:")
       THEN lrecl := buf.Length;
    Image :- buf;
    IF  f_spec NE "TTY:"
        THEN BEGIN
                IF f == NONE
                       THEN f :- NEW Infile(ddname);
                f.Open(Image)
              END;
    Image.Setpos(Image.Length+1);
    f_opened := TRUE;
exit:
END of Open;
```

```
PROCEDURE Close;
BEGIN
    IF NOT f_opened
         THEN BEGIN
                 message :- conc2("[% in_file.Close: File ´",f_spec);
                 message :- conc2(message,"´ is already closed.]");
                 vpimoi(message);
                 message :- Copy("[Execution continues.]");
                 vpimoi(message);
              END;
    IF f_spec NE "TTY:"
         THEN f.Close;
    f_opened := FALSE
END of Close;


INTEGER PROCEDURE term_type;
    term_type := t_type;


BOOLEAN PROCEDURE Endfile;
    Endfile :=  IF f_spec = "TTY:"
                    THEN (NOT f_opened) OR f_endfile
                    ELSE f.Endfile;
```

```
PROCEDURE Inimage;
BEGIN
    TEXT temp;

    IF NOT f_opened
         THEN BEGIN
                 message :- conc2("? in_file.Inimage: File ´",f_spec);
                 message :- conc2(message,"´ is closed.");
                 vpimoi(message);
                 IF booleanrequest("Do you want to open the file?",
                                        TRUE,TRUE)
                    THEN BEGIN
                          IF f_spec = "TTY:"
                                 THEN BEGIN
                                         Image :- Blanks(255);
                                         f_opened := TRUE
                                      END
                                 ELSE BEGIN
                                         f.Open(Blanks(lrecl));
                                         f_opened := TRUE
                                   END
                         END
                    ELSE Abort("Attempting to read closed file.");
                END;
```

```
IF f_spec = "TTY:"
     THEN BEGIN
             Image :- Blanks(Image.Length);
             vpiini(Image);
             Image.Setpos(1);
            IF (IF Image.Strip =/= NOTEXT
                    THEN Image.Strip = "/*"
                    ELSE FALSE)
                 THEN f_endfile := TRUE;
          END
     ELSE BEGIN
             f.Inimage;
             IF echo AND NOT f.Endfile
                 THEN BEGIN
                          temp :- f.Image.Strip;
                          vpimoi(temp)
                      END;
             IF divert
                    THEN BEGIN
                            TEXT temp;
                            IF (IF Image.Strip =/= NOTEXT
                                    THEN Image.Strip = "/*"
                                    ELSE FALSE)
                            THEN BEGIN
                                    f.Close;
                                    echo := false;
                                    message :- conc2("[% in_file.Inimage: Input from file ´",f_spec);

                                    message :- conc2(message,"´ diverted to tty:.]");
                                    vpimoi(message);
                                    f_spec :- Copy("TTY:");
                                    lrecl := 255;
                                    Image :- Blanks(255);
                                    vpiini(Image);
                                    IF (IF Image.Strip =/= NOTEXT
                                            THEN Image.Strip = "/*"
                                            ELSE FALSE)
                                        THEN f_endfile := TRUE;
                                   END
                             ELSE BEGIN
                                    temp :- Copy(Image.Strip);
                                    vpimoi(temp)
                                  END;
                       END;
```

```
                END;
    Image.Setpos(1);
    temp :- Image.Strip;
    INSPECT log DO
    BEGIN
        Outtext(temp);
        Outimage
    END;
END of Inimage;
```

```
BOOLEAN PROCEDURE Lastitem;
    IF Endfile
        THEN Lastitem := TRUE
        ELSE BEGIN
                WHILE Image.Sub(Image.Pos,Length+1-Image.Pos).Strip = NOTEXT
                                AND NOT Endfile DO
                BEGIN
                    IF f_spec = "TTY:"
                        THEN BEGIN
                                message :- Copy("# ");
                                message.Setpos(2);
                                vpiboi(message)
                             END;
                    Inimage
                END;

                IF Endfile
                    THEN Lastitem := TRUE
                    ELSE BEGIN
                                Lastitem := FALSE;
                                IF Image.Sub(1,1) = " "
                                    THEN BEGIN
                                             WHILE Image.Getchar = ' ' DO;
                                             Image.Setpos(Image.Pos-1)
                                         END;
                             END
             END of Lastitem;
```

```
TEXT PROCEDURE Intext(w); INTEGER w;
BEGIN
    check_open("Intext");
    IF Image.Pos + w <= Image.Length
        THEN BEGIN
                Intext :- Copy(Image.Sub(Image.Pos,w));
                Image.Setpos(Image.Pos+w)
             END
        ELSE BEGIN
                TEXT temp;
                IF Image.More
                    THEN BEGIN
                                temp :- conc2(" ",Image.Sub(Image.Pos,Image.Length+1-Image.Pos));
                                w := w - (Image.Length+1 - Image.Pos)
                            END
                    ELSE temp :- Blanks(1);
                Inimage;
                IF f_spec = "TTY:"
                    THEN BEGIN
                                message :- Copy("# ");
                                message.Setpos(2);
                                vpiboi(message)
                         END;
                WHILE w >= Image.Length and Image.Sub(1,2) NE "/*" DO
                BEGIN
                    temp :- conc2(temp,Image);
                    w := w - Image.Length;
                    IF f_spec = "TTY:"
                        THEN BEGIN
                                message :- Copy("# ");
                                message.Setpos(2);
                                vpiboi(message)
                            END;
                    Inimage
                END;
                IF Image.Sub(1,2) = "/*"
                    THEN eof_abort("Intext",f_spec);
                IF w > 0
                    THEN BEGIN
                                Inimage;
                                IF f_spec = "TTY:"
                                    THEN BEGIN
                                                message    Copy( # ");
```

```
                                             message.Setpos(2);
                                             vpiboi(message)
                                         END;
                                IF Image.Sub(1,2) = "/*"
                                THEN eof_abort("Intext",f_spec);
                                temp :- conc2(temp,Image.Sub(1,w));
                                Image.Setpos(w+1)
                            END;
                Intext :- Copy(temp.Sub(2,w))
             END;
END of Intext;
```

```
INTEGER PROCEDURE Inint;
BEGIN
    TEXT temp;
    IF Lastitem
         THEN eof_abort("Inint",f_spec);
    temp :- Image.Sub(Image.Pos,Image.Length-Pos);
    Inint := temp.Getint;
    Image.Setpos(Image.Pos+temp.Pos-1)
END of Inint;


REAL PROCEDURE Infrac;
BEGIN
    TEXT temp;
    IF Lastitem
         THEN eof_abort("Infrac",f_spec);
    temp :- Image.Sub(Image.Pos,Image.Length-Pos);
    Infrac := temp.Getfrac;
    Image.Setpos(Image.Pos+temp.Pos-1)
END of Infrac;

REAL PROCEDURE Inreal;
BEGIN
    TEXT temp;
    IF Lastitem
         THEN eof_abort("Inreal",f_spec);
    temp :- Image.Sub(Image.Pos,Image.Length-Pos);
    Inreal := temp.Getint;
    Image.Setpos(image.Pos+temp.Pos-1)
END of Inreal;
```

```
CHARACTER PROCEDURE Inchar;
BEGIN
    check_open("Inchar");
    IF Image.More
        THEN Inchar := Image.Getchar
        ELSE BEGIN
                IF f_spec = "TTY:"
                    THEN BEGIN
                            message :- Copy("# ");
                            message.Setpos(2);
                            vpiboi(message)
                        END;
                Inimage;
                IF Image.Sub(1,2) = "/*"
                    THEN eof_abort("Inchar",f_spec)
                    ELSE Inchar := Image.Getchar
             END
END of Inchar;

lrecl := 255;
f_spec  :- upcase(frontstrip(f_spec.Strip));
IF f_spec NE "TTY:"
    THEN BEGIN
                 f_spec :- conc2(f_spec," ");
                 ddname :- getddinput(f_spec,lrecl);
                 f :- NEW Infile(ddname);
                 echo := FALSE
          END;

END of Infile;
```

```
CLASS out_file(f_spec); VALUE f_spec; TEXT f_spec;

PROTECTED        f_spec,
                 f,
                 f_opened,
                 t_type,
                 check_open,
                 vpiboi,
                 vpimoi;

BEGIN


REF(Outfile) f;

TEXT Image,Message;

BOOLEAN f_opened,
        echo;

INTEGER t_type;

COMMENT t_type = 0 <=> ASCII terminal or file
        t_type = 1 <=> key-paired APL terminal or file
        t_type = 2 <=> bit-paired APL terminal or file
        ;

EXTERNAL assembly PROCEDURE vpiboi,
                            vpimoi,
                             vpiabort;

EXTERNAL TEXT PROCEDURE convtoapl,
                        getddoutput,
                        upcase,
                        frontstrip,
                        conc2;

EXTERNAL INTEGER PROCEDURE      integerrequest;

EXTERNAL BOOLEAN PROCEDURE      booleanrequest;
```

```
PROCEDURE check_open(caller); VALUE caller; TEXT caller;
    IF NOT f_opened
        THEN BEGIN
                 message :- conc2("% out_file.",caller);
                 message :- conc2(message,": File ´");
                 message :- conc2(message,f_spec);
                 message :- conc2(message,"´ is closed.");
                 vpimoi(message);
                IF booleanrequest("Do you want to open this file?",
                                        TRUE, TRUE)
                   THEN Open(Blanks(integerrequest(
                               "Enter length of output lines: ",
                               80, 10, 800,TRUE)))
                    ELSE Abort ("Attempt to access closed file.")
             END;

BOOLEAN PROCEDURE opened;
    opened := f_opened;

TEXT PROCEDURE file_spec;
    file_spec :- Copy(f_spec);

PROCEDURE Setpos(i); INTEGER i;
BEGIN
    check_open("Setpos");
    Image.Setpos(i);
END of Setpos;

INTEGER PROCEDURE Pos;
BEGIN
    check_open("Pos");
    Pos := Image.Pos;
END of Pos;

BOOLEAN PROCEDURE More;
BEGIN
    check_open("More");
    More := Image.More;
END of More;

INTEGER PROCEDURE Length;
BEGIN
    check_open("Length");
    Length := Image.Length;
END of Length;
```

```
PROCEDURE Open(buf); TEXT buf;
BEGIN
    IF f_opened
       THEN BEGIN
                message :- conc2("? out_file.Open: File ´",f_spec);
                message :- conc2(message,"´ is already open.");
                vpimoi(message);
               IF booleanrequest("Do you want to close the file? ",
                                        TRUE, TRUE)
                   THEN BEGIN Close; Open(buf) END
                    ELSE Abort ("Attempt to open an open file.");
            END;
    Image :- buf;
    IF f_spec NE "TTY:"
       THEN BEGIN
               IF f == NONE
                  THEN f :- NEW Outfile(getddoutput(
                                 f_spec,buf.Length));
               f.Open(Image)
            END;
    f_opened := TRUE;
END of Open;

PROCEDURE Close;
BEGIN
    IF NOT f_opened
       THEN BEGIN
                message :- conc2("[% out_file.Close: File ´",f_spec);
                message :- conc2("´ is already closed.]");
                vpimoi(message);
                message :- copy("[Execution continues.]");
                vpimoi(message)
            END;
    IF f_spec NE "TTY:"
       THEN f.Close;
    f_opened := FALSE
END of Close;
```

```
PROCEDURE Outtext(t); VALUE t; TEXT t;
BEGIN
    check_open("Outtext");

    IF t.Length + Image.Pos <= Image.Length + 1
        THEN BEGIN
                TEXT temp;
                temp :- Image.Sub(Image.Pos,t.Length);
                Image.Setpos(Image.Pos + t.Length);
                temp := t
             END
        ELSE BEGIN
                INTEGER cnt;
                Breakoutimage;
                cnt := 1;
                WHILE t.Length - cnt >= Image.Length DO
                BEGIN
                    Image :- t.Sub(cnt,Image.Length);
                    Breakoutimage;
                    cnt := cnt + Image.Length
                END;
                IF cnt <= t.Length
                        THEN BEGIN
                                TEXT temp;
                                temp :- Image.Sub(1,
                                        (t.Length-cnt)+1);
                                temp := t.Sub(cnt,
                                        (t.Length-cnt)+1);
                                Breakoutimage
                             END;
                    Outimage
             END
END of Outtext;
```

```
PROCEDURE Outchar(c); CHARACTER c;
BEGIN
    check_open("Outchar");
    IF NOT Image.More
        THEN Breakoutimage;
    Image.Putchar(c);
END of Putchar;

PROCEDURE Outint(i,w); INTEGER i, w;
BEGIN
    TEXT temp;
    INTEGER t;
    check_open("Outint");
    IF Image.Pos + w > Image.Length
        THEN Breakoutimage;
    t := Image.Pos;
    temp :- Image.Sub(t,w);
    temp.Putint(i);
    Image.Setpos(t+w)
END of Outint;

PROCEDURE Outfrac(i,n,w); INTEGER i, n, w;
BEGIN
    TEXT temp;
    INTEGER t;
    check_open("Outfrac");
    IF Image.Pos + w > Image.Length
        THEN Breakoutimage;
    t := Image.Pos;
    temp :- Image.Sub(t,w);
    temp.Putfrac(i,n);
    Image.Setpos(t+w)
END of Outfrac;

PROCEDURE Outreal(p,n,w); REAL p; INTEGER n, w;
BEGIN
    TEXT temp;
    INTEGER t;
    check_open("Outreal");
    IF Image.Pos + w > Image.Length
        THEN Breakoutimage;
    t := Image.Pos;
    temp :- Image.Sub(t,w);
    temp.Putreal(p,n);
    Image.Setpos(t+w);
END of Outreal;

PROCEDURE Outfix(p,m,w); REAL p; INTEGER m, w;
BEGIN
    TEXT temp;
    INTEGER t;
    check_open("Outfix");
    IF Image.Pos + w > Image.Length
        THEN Breakoutimage;
    t := Image.Pos;
    temp :- Image.Sub(t,w);
    temp.Putfix(p,m);
    Image.Setpos(t + w)
END of Outfix;
```

```
    PROCEDURE set_ascii;
    BEGIN
        check_open("set_ascii");
        IF Image.Strip NE NOTEXT
            THEN Outimage;
        IF t_type > 0
            THEN BEGIN
                      Outchar(Char(15)); ! Char(15) = <si>;
                      Breakoutimage;
                  END;
        t_type := 0
    END of set_ascii;


    PROCEDURE set_key_paired;
    BEGIN
        check_open("set_key_paired");
        IF Image.Strip NE NOTEXT
            THEN Outimage;
        IF t_type = 0
            THEN BEGIN
                      Outchar(Char(14)); ! Char(14) = <so>;
                      Breakoutimage
                  END;
        t_type := 1
    END of set_key_paired;


    PROCEDURE set_bit_paired;
    BEGIN
        check_open("set_bit_paired");
        IF Image.Strip NE NOTEXT
            THEN Outimage;
        IF t_type = 0
            THEN BEGIN
                      Outchar(Char(15)); ! Char(15) = <si>;
                      Breakoutimage
                  END;
        t_type := 2;
    END of set_bit_paired;


    INTEGER PROCEDURE term_type;
    BEGIN
        check_open("term_type");
        term_type := t_type;
    END of term_type;
```

```
PROCEDURE Outimage;
BEGIN
    check_open("Outimage");
    IF t_type = 0
        THEN BEGIN
                IF f_spec = "TTY:"
                    THEN BEGIN
                             vpimoi(Image);
                             Image :- Blanks(Image.Length);
                         END
                    ELSE IF echo
                            THEN BEGIN
                                     vpimoi(Image);
                                     f.Outimage
                                 END
                             ELSE f.Outimage
             END
        ELSE BEGIN
                TEXT holder;
                holder :- Image;
                Image :- Blanks(Image.Length);
                Image := convtoapl(holder,t_type);
                IF f_spec = "TTY:"
                    THEN BEGIN
                             vpimoi(Image);
                             Image :- Blanks(Image.Length);
                         END
                    ELSE IF echo
                            THEN BEGIN
                                    vpimoi(Image);
                                    f.Outimage
                                 END
                            ELSE f.Outimage
             END;
    Image.Setpos(1)
END of Outimage;


PROCEDURE apl_outimage;
BEGIN
    check_open("apl_outimage");
    IF f_spec = "TTY:"
```

```
        THEN BEGIN
                vpimoi(Image);
                Image :- Blanks(Image.Length);
             END
        ELSE IF echo
                THEN BEGIN
                        vpimoi(Image);
                        f.Outimage
                     END
                ELSE f.Outimage;
    Image.Setpos(1)
END of apl_outimage;
```

```
PROCEDURE Breakoutimage;
BEGIN
    check_open("Breakoutimage");
    IF t_type = 0
        THEN BEGIN
                IF f_spec = "TTY:"
                    THEN BEGIN
                             vpiboi(Image);
                             Image :- Blanks(Image.Length);
                         END
                    ELSE IF echo
                            THEN BEGIN
                                    vpiboi(Image);
                                    f.Outimage
                                 END
                            ELSE f.Outimage
             END
        ELSE BEGIN
                TEXT holder;
                holder :- Image;
                Image :- Blanks(Image.Length);
                Image := convtoapl(holder,t_type);
                IF f_spec = "TTY:"
                    THEN BEGIN
                             vpiboi(Image);
                             Image :- Blanks(Image.Length);
                         END
                    ELSE IF echo
                            THEN BEGIN
                                     vpiboi(Image);
                                     f.Outimage
                                 END
                            ELSE f.Outimage
             END;
    Image.Setpos(1)
END of Breakoutimage;


PROCEDURE apl_breakoutimage;
BEGIN
    check_open("apl_breakoutimage");
    IF f_spec = "TTY:"
```

```
        THEN BEGIN
                vpiboi(Image);
                Image :- Blanks(Image.Length);
             END
        ELSE IF echo
                THEN BEGIN
                        vpiboi(Image);
                        f.Outimage
                     END
                ELSE f.Outimage;
    Image.Setpos(1)
END of apl_breakoutimage;
```

```
COMMENT Initialization;

    f_spec :- upcase(frontstrip(f_spec.Strip));
    IF f_spec NE "TTY:"
       THEN f_spec :- conc2(f_spec," ");
    t_type := 0


END of out_file;
```

```
    REF(in_file) Sysin;
    REF(out_file) Sysout;

    vpisave;
    initial;
    Sysin :- NEW in_file("TTY:");
    Sysin.Open(Blanks(255));
    Sysout :- NEW out_file("TTY:");
    Sysout.Open(Blanks(255));

    INSPECT Sysin DO
    INSPECT Sysout DO
%SOURCE
```

The contents of file POSTFIX SIMULA is the following text.

```
     Sysin.Close:
     Sysout.Close;
     vpirest
END
```